Report Number: WUCSE-2003-72

2003-11-07

Context-Sensitive Binding: Flexible Programming Using Transparent Context Maintenance

Rohan Sen and Gruia-Catalin Roman

Context-aware computing is a new paradigm whose emergence has been fostered by a growing reliance on light and mobile computing devices, which adapt their behavior to changing environmental conditions. The dynamic nature of the environment is a direct result of the mobility of people and devices. Because the development of applications that entail a significant level of dynamic adaptation is a difficult and error-prone task, new design methods and constructs are needed. Precise and flexible specification of the resources needed to operate in specific contexts combined with transparent context management can simplify the development process. In this paper we... **Read complete abstract on page 2.**

Context-Sensitive Binding: Flexible Programming Using Transparent Context Maintenance

Rohan Sen and Gruia-Catalin Roman

Complete Abstract:

Context-aware computing is a new paradigm whose emergence has been fostered by a growing reliance on light and mobile computing devices, which adapt their behavior to changing environmental conditions. The dynamic nature of the environment is a direct result of the mobility of people and devices. Because the development of applications that entail a significant level of dynamic adaptation is a difficult and error-prone task, new design methods and constructs are needed. Precise and flexible specification of the resources needed to operate in specific contexts combined with transparent context management can simplify the development process. In this paper we propose a particular embodiment of this general design strategy in the form of a novel programming construct called context-sensitive binding. The approach allows programmers to define and use in their programs objects whose behavior is supported by code discovered at runtime within the computing environment surrounding the de-vice. The binding between the object in the program and the support object that delivers its realization is maintained transparently and is altered as the environment changes, thus making the binding context sensitive. The criteria for choosing among viable support objects are prescribed at the time the object is first instantiated. The paper introduces the concept of context sensitive binding, describes a Java-based implementation, and explores the programming implications of the proposed construct.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-72

Context-Sensitive Binding: Flexible Programming Using Transparent Context Maintenance

Authors: Sen, Rohan; Roman, Gruia-Catalin

November 7, 2003

Abstract: Context-aware computing is a new paradigm whose emergence has been fostered by a growing reliance on light and mobile computing devices, which adapt their behavior to changing environmental conditions. The dynamic nature of the environment is a direct result of the mobility of people and devices. Because the development of applications that entail a significant level of dynamic adaptation is a difficult and error-prone task, new design methods and constructs are needed. Precise and flexible specification of the resources needed to operate in specific contexts combined with transparent context management can simplify the development process. In this paper we propose a particular embodiment of this general design strategy in the form of a novel programming construct called context-sensitive binding. The approach allows programmers to define and use in their programs objects whose behavior is supported by code discovered at runtime within the computing environment surrounding the device. The

Context-Sensitive Binding: Flexible Programming Using Transparent Context Maintenance

Rohan Sen and Gruia-Catalin Roman

Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{rohan.sen, roman}@wustl.edu

Abstract. Context-aware computing is a new paradigm whose emergence has been fostered by a growing reliance on light and mobile computing devices, which adapt their behavior to changing environmental conditions. The dynamic nature of the environment is a direct result of the mobility of people and devices. Because the development of applications that entail a significant level of dynamic adaptation is a difficult and error-prone task, new design methods and constructs are needed. Precise and flexible specification of the resources needed to operate in specific contexts combined with transparent context management can simplify the development process. In this paper we propose a particular embodiment of this general design strategy in the form of a novel programming construct called context-sensitive binding. The approach allows programmers to define and use in their programs objects whose behavior is supported by code discovered at runtime within the computing environment surrounding the device. The binding between the object in the program and the support object that delivers its realization is maintained transparently and is altered as the environment changes, thus making the binding context sensitive. The criteria for choosing among viable support objects are prescribed at the time the object is first instantiated. The paper introduces the concept of context sensitive binding, describes a Java-based implementation, and explores the programming implications of the proposed construct.

1 Introduction

The growing dependence on light, portable computing devices and continuing improvements in wireless communication technology have contributed to the increasing popularity of mobile computing. Mobile computing is characterized by the physical mobility of hosts which results in frequent disconnections, transient interactions and decoupled computing. The physical mobility of hosts has fostered new and interesting patterns of interactions, heretofore not seen in the computing world. Nomadic computing, peer to peer communication, hoarding of resources prior to disconnection and delegation of tasks are suggestive of the new ways physically mobile devices interact with each other. The unique characteristics of mobile devices have fuelled the need for a new class of applications

that can *adapt* dynamically to frequent and unpredictable changes in their operational environment.

The idea of applications that adapt to their operational environment is not new. The notion of dynamic binding in C++ [1] deferred the decision of binding to an object till runtime. However, these decisions were one time events which were based on the current state of the operational environment and were irreversible for the duration of execution of the program. This level of adaptation is not sufficient for mobile computing, where devices encounter an eclectic set of environments. Not only should an application be able to defer its decision to bind to an object at runtime, it must also be able to frequently revisit this decision and modify it based on changes in its operational environment. The decision process in mobile computing is further complicated due to the fact that, by virtue of its wireless communication capability, an application may be able to bind to objects that reside on a set of remote hosts that are reachable, but where the set of such reachable hosts changes over time. Designing applications that achieve such a level of adaptability requires significant programming overhead in the form of managing concurrency, designing customized protocols, and bookkeeping. The effort is significant, even for experienced programmers.

Powerful programming constructs which abstract the low level details of communication, concurrency, and adaptability can reduce the complexity of programming adaptable applications to a level that is comprehensible by the average programmer. Our goal is to leverage the power of such constructs to transform a simple, static program into a adaptable, dynamic program without significantly increasing the programming effort. Examples of such constructs are those which allow access to remote objects as if they were local, transparent dynamic binding and re-binding to remote objects, and delegation of tasks between objects by employing policy based decision making. Using such constructs in a correct manner can facilitate rapid development of adaptable applications.

In this paper, we introduce *context-sensitive binding*, a novel concept that decouples the interface of an object from its implementation to a degree and in a manner never attempted before. The binding between the interface and some implementation of the object is dynamic and maintained transparently in the face of changes in operational environment. This allows a one time specification of interface by the programmer and multiple realizations of the object, each customized to its operational environment. Programmer specified *policies* determine where the realization of the interface comes from. Context-sensitive binding allows the application programmer to interact with the interface as if it were a local object even though the realization of the interface may come from one or more remote objects. The policies specified by the programmer are used to build a window on its operational environment or *context*, which is a set of reachable entities around the host device. The window is a subset of the context and contains neighboring hosts which have properties that are of interest to the application. The support object that delivers the realization of a programmer specified interface (hereinafter referred to as the *provider*) is chosen from among the hosts that fall within the constructed window. This ensures that the

most relevant realizations of a given interface are considered among accessible alternatives. Additional flexibility and strength is achieved by including support for migration of partially computed tasks between different providers as well as optimizations to minimize losses due to failure and migration of tasks between providers.

In order to demonstrate the feasibility of context-sensitive binding, we present a particular embodiment of the concept in the form of middleware implemented in Java. We offer the programmer a set of constructs to create interfaces to objects whose actual realizations come from remote providers that are within the current context of the host device. We provide middleware that builds and manages contexts in a manner that is transparent to the programmer and incorporates optimizations such as periodic caching of partial computations. The application programmer is thus able to specify a set of requirements and/or a certain class of devices the application needs in order to be able to carry out its task. The middleware creates a window of interest within the host device's context and discovers eligible providers within the context. It binds to the *best* candidate within the context, where *best* is defined by a set of predefined metrics. It maintains this binding transparently, even if the provider of the realization of the resource changes, and returns complete results of the computation as if it were coming from the local interface.

The remainder of the paper is organized as follows: Section 2 highlights the core concept of context-sensitive binding and illustrates them via the use of a high level example. Section 3 describes the assumptions, design and a Java implementation of middleware that demonstrates the feasibility of context-sensitive binding. We discuss context-sensitive bindings from an application programmer perspective in Section 4. In Section 5, we reflect on outstanding research issues and discuss the applicability of such middleware in a variety of settings. We draw conclusions in Section 6.

2 The Context-Sensitive Binding Concept

Developing dependable and adaptable applications for mobile devices is expensive because applications that reside on mobile devices have to be engineered to be robust enough so as to not fail under a wide range of circumstances that could occur during operation, yet be flexible enough to allow interaction with the heterogenous set of hosts that it may encounter. Facilitating rapid development of adaptable applications in settings characterized by mobility, transient interactions, and disconnection is a complex task. This is where context-sensitive binding can help.

2.1 The Core Concept

Context-sensitive binding is a design strategy that enables rapid development of adaptable context-aware applications by decoupling the object interface from its implementation code or realization. Based on requirements, the programmer

specifies an interface which contains a list of methods for which he needs an implementation. Along with this list of methods, he specifies a set of external constraints which determine the sources from where the implementation may be obtained. The list of methods and constraints together form a *policy*. The policy is passed to the system which applies the policy on a *registry* or set of reachable providers (entities which provide realizations of methods or entire classes). The set of reachable providers is filtered to give a set of providers that implement all the required methods, i.e., the list of methods in the programmer specified interface must be a subset of the list of methods of the provider. The filtered set, consisting of only those providers that implement the required methods is filtered again based on the external constraints provided by the programmer. The result is an *eligibility set*, a window on the context of the device consisting of providers that meet the requirements set forth in the policy. The system uses predetermined metrics to choose the best provider and forms a binding between the programmer specified interface and the chosen provider. In the case of disconnection, the system automatically rebinds to the next best provider. Context-sensitive binding boasts several novel features that distinguish it from prior work:

- *Transparent Maintenance of Binding* - The binding between the interface specified by the programmer and the object that supports the realization of the interface (*provider)* is maintained in a transparent manner. The programmer interacts only with the local interface and calls methods on that interface. The interface delegates the call to the provider. Choosing providers and switching between providers are hidden from the programmer.
- *Policy Based Choosing* - The set of qualifying providers that represent realization of an interface is chosen based on programmer specified policies.
- *Metric Based Evaluation* - The best provider from the set of all qualifying providers is chosen based on predetermined metrics.
- *Continuous Binding for Ad hoc Mobile Environments* - Context-sensitive binding provides for continuous binding between the programmer specified interface and the provider in the face of disconnection or in search of a better quality of service. The provider for a given interface is replaced transparently by another provider if the former provider gets disconnected or does not meet quality of service criteria.

Context-sensitive binding is targeted towards ad hoc networks where frequent disconnections and transient interactions are not conducive to static bindings. While dynamic and transparent nature of context-sensitive binding facilitates development of applications for ad hoc networks, its applicability is not restricted to ad hoc networks alone. Context sensitive binding can be used in any setting where it is useful to decouple the object interface from the implementation and different realizations of the interface are selected in accordance with some programmer supplied selection policy.

2.2 Expanding the Concept - An Operational Perspective

The notion of dynamic binding is tied into the decision process by which a specific piece of code is selected from a set of candidates at runtime to fulfil a given task. While context-sensitive binding shares some commonality with other approaches to dynamic binding (The idea of making binding decisions at runtime), the novelty it brings lies in the fact that it does not make this binding decision once but on a continuous basis in direct response to changes in the operational environment. The key to making correct binding decisions rests on the ability to ensure effective building, maintenance and management of eligibility sets within the context of the host device. The remainder of this subsection is devoted to descriptions of how eligibility sets are defined, built and evaluated on a constant basis to facilitate correct binding decisions.

Defining Eligibility Sets

In context-sensitive binding, an eligibility set is defined for each interface-implementation pair. The first step towards building eligibility sets is to define what is of interest. This specification is done using *policies*. A policy has two components - an endogenous component and an exogenous component. The endogenous component is a series of constraints that are determined by the programmer specified interface for which the eligibility set is being defined. These endogenous constraints consist of the fields and method signatures of the interface and constrain the eligibility set to only those objects that provide implementations for the required methods. The exogenous component contains a further set of external constraints that shrink the set further. These external constraints can be constraints on any variable that is not internal to the specified interface. Examples of external constraints can be location of the provider, relative velocity of the provider, quality of service parameters and security clearances among others. The eligibility set for a given interface is defined as the subset of the context of the host device that obeys all the endogenous and exogenous constraints specified in the policy associated with the interface.

Building Eligibility Sets

Given a policy that defines an eligibility set for a given interface, the aim is to build up a set of objects that satisfy the constraints specified in the policy. The set of reachable hosts that make up the context can be established by sending out beacons and collecting responses. Once a set of such hosts is established, they are filtered according to their capabilities, i.e., the endogenous constraints are applied. After the set has been filtered to include only those providers who satisfy the endogenous constraints, the exogenous constraints are applied to filter the set farther. The result is the eligibility set.

It should be noted here that our rationale for applying endogenous constraints first is that we did not want to eliminate any provider that had the required capabilities but fell short due to exogenous constraints which are dynamic and constantly changing. In other words, we did not want to penalize a provider with non-inclusion in our eligibility set simply because it did not satisfy

some external constraint like location, velocity etc at a given point in time. By deferring filtering based on exogenous constraints to the second round, we give every provider a fair chance to be selected on that basis of its capabilities alone.

Evaluation and Eligibility Set Management

Once an eligibility set has been built, there are two remaining tasks: (1) evaluating all the providers in the eligibility set to choose the most suitable candidate and (2) keeping the contents of the eligibility set up to date as the context of the host device changes.

Evaluation of the providers in the eligibility set is done using a metric. The metric of choice that is used to evaluate providers in the eligibility set is volatility. The *volatility* of an provider is defined as the probability that it will become unreachable within a set interval of time. The current location and velocity of an provider is used to determine the relative direction of travel of the entity, and the safe distance [2] is used to calculate the volatility figure. It is assumed that all providers publicly broadcast their location and velocity. The provider with the lowest volatility is considered best and is the primary candidate for binding. Should this candidate become unavailable at some point in the future, the interface is automatically rebound to the best candidate available at that time. It should be observed that volatility is suggested as the metric of choice for evaluating providers in the eligibility set because we consider spatial orientation and physical mobility to be the chief causes of changes in context. However, if other factors drive changes in context, an alternate metric can be used to determine the suitability of a candidate provider. An example of this is differing levels of security clearances as a metric when browsing classified data.

The remaining task is that of keeping the eligibility set up to date as the context of the host device changes. Once again, the volatility metric is used for this task. Queries are issued at certain time intervals to ensure that a provider is still reachable. The time interval is determined by the volatility of the provider so more volatile providers are queried more often than less volatile providers. If no response is received, the provider is removed from the eligibility set. In addition to maintaining providers already on the eligibility set, periodic rebroadcasts of the endogenous and exogenous constraints capture newly reachable providers and add them to the eligibility set by the procedure detailed above.

Ancillaries Supporting Optimizations

In addition to the basic functions that build an eligibility set and choose a suitable provider from within, we offer ancillaries that further optimize the core concept. We describe two mechanisms: (1) periodic capturing of partial computations and (2) commissioning of providers to finish incomplete tasks.

As the context of the host device changes, the chosen provider may become unreachable or unable to satisfy the exogenous requirements. At such a point, a new provider must be chosen and the interface rebound to the new provider. When this happens, the computation occurring on the former provider is suspended and restarted on the latter provider. Frequent restarts of the computation

can result in losses of results and may increase the time taken for the computation by a significant amount. To mitigate this problem, the state of the partial computation is cached periodically and when there is a switch between providers, the cached state is pushed to the new provider to enable resumption rather than restarting of the computation. The periodicity of the state saving is determined, once again, by the volatility of the provider with more stable providers sending back partial results less frequently.

Another mechanism for optimization of the computation is *commissioning*. This mechanism assumes a significant amount of knowledge exchange between providers. The concept is essentially this: if a provider knows it is going to become unreachable in the near future and also knows of an alternate provider that has the same characteristics as itself, it can *commission* the other provider to finish its task. This results in an increased efficiency because the time to look for another suitable provider is saved and the handover between providers can happen in a more orderly fashion so as to minimize the losses due to the switch.

2.3 Illustrative Example

Before concluding this section, we present a simple example designed to explore a setting in which these concepts may be used. Consider a miner who must carry out frequent inspections of an underground mine shaft. As he moves through the mine, he would like the lights around him to turn on and stay on until he leaves the area. To do this manually, he would have to turn on and off many switches as he moved through the mine. However, this particular mine is equipped with switches at 100 meter intervals than can respond to commands issued on a PDA and transmitted over a wireless ad hoc connection to the switches. The miner uses a simple program and specifies his interest in light switches and the requirement that he would like any such switch within 100 meters of his position to be set to the "on" position. As he enters the mine, he starts his program which discovers a set of switches in its context. According to his policy, only light switches are considered. The program binds to the light switch and turns it on. As the miner walks forward, the first switch moves out of the context of the miner's PDA, i.e., it becomes unreachable from the PDA. Since the switch is not unreachable, it reverts back to its default state, which is the "off" position. However, the second switch, which was initially unreachable now falls within the context and turns on in response to directives from the PDA. This process repeats as the miner walks through the mine. The miner can thus use a single conceptual switch on his PDA to control a changing set of light switches as he moves through the mine.

In the next section, we demonstrate the feasibility of context-sensitive binding by describing a design and implementation of a middleware that captures our strategy for realization of the context-sensitive binding concept.

3 Middleware Design

In this section, we describe a middleware implemented in Java that demonstrates the feasibility of the context-sensitive binding concept. It represents one of many possible designs that realize the general concept. This design is targeted towards wireless ad hoc networks which exhibit transient interactions and decoupled computing due to physical mobility of hosts and the logical mobility of agents. We begin by identifying a set of assumptions that we made in the design of the middleware. The presentation continues with an outline of the middleware design. The runtime behavior is illustrated by examining a trace through a sample execution. Finally, we provide selected implementation details for a more in depth understanding of the system.

3.1 Assumptions

The middleware we describe in this section is intended to demonstrate the feasibility of the concept. As such, we make a set of assumptions that make the middleware development simpler but do not detract from illustrating the richness of the concept. Designing a system where we can relax these assumptions is part of our planned future work which is described in a later section.

General Assumptions

- The system is intended for use in wireless ad hoc networks.
- All participants in the system are assumed to carry code written in Java.
- There is an external coordination mechanism that handles the discovery of and communication with providers.
- The primary cause for a change in a device's context is physical mobility, i.e., a change in physical location may cause a change in context.
- All participants in the system provide their location and velocity in a standard format that is interpretable by all.
- The *volatility* of an entity is the metric used by default to determine the most suitable of qualifying providers. Alternate metrics may be plugged into the system as required.

In addition to the general assumptions listed above, we assume that a policy consisting of endogenous and exogenous constraints is unique enough that searching for providers who obey that policy will not generate any false positives. We also assume that providers do not advertise spurious capabilities and that the semantics of a capability are the same across all potential participants in the community. Finally, we assume that a programmer never advertises a policy with constraints that are in conflict.

3.2 System Design

In this subsection, we outline the design of a middleware that supports context-sensitive binding. We begin by describing the general architecture of the system using a layered approach. We follow that by a discussion of the main components

of the system. We conclude with a trace through the middleware to show how the various components interact with each other.

General System Architecture

The discussion of the general system architecture is split into two parts - (1) the architecture for the programmer specified interface (hereinafter referred to as the program) and (2) the architecture of the provider. We use the layered approach to show how these components interact with the middleware as well as with each other.

The topmost layer on the program side is the user application that makes use of context-sensitive binding. It consists of static classes, which are standard Java classes and *dynamic* classes, which use context-sensitive binding. The application issues method calls to the dynamic classes in the same manner that it does to static classes. However, the dynamic classes are simply skeletons, i.e., lists of methods and their parameters. They do not contain any implementation. When a dynamic class is created, the application programmer specifies a policy that determines where the implementation or realization comes from. To obtain the implementation, the dynamic class delegates the task to the next layer, which is the generic interface to the context-sensitive binding middleware. The generic interface packages the method call and its accompanying policy as a standard object and passes it down to the next layer, which is the context management layer.

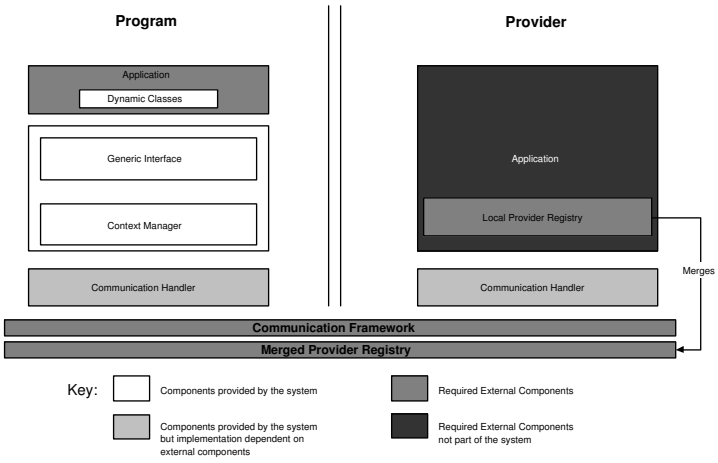

Fig. 1. Layer Diagram of the System

The context management layer uses the policy to build an eligibility set that is applicable to the dynamic class in question. It does this by broadcasting a query consisting of the constraints in the policy to a registry of reachable providers.

The communication occurs through an external coordination framework which we do not provide as part of the system. Communication between the context manager and the communication layer is handled by a communication handler that can be written to interface with the communication layer of choice. Once the context manager receives replies to its query, it picks the provider that is best suited to service the request by using the *volatility* metric. It then delegates the request to that particular provider by sending a message via the communication layer. Further interactions between the program and provider are also handled by the context manager. Replies from the provider are propagated up to the generic interface which forwards them to the dynamic class within the application. An illustration of this can be seen in Fig. 1.

On the provider side, the hierarchy is less complex. There are just two layers. The application layer represents the functionality of the provider, i.e., it is the program that processes the jobs that are delegated to it. The other layer is the communication handler which sits below the application layer and simply handles all communication details, similar to its counterpart on the program side. The illustration of this can be seen in Fig. 1.

Characterizing the Components

Having described the system architecture on a high level, we now describe the main components of the system. The system is designed in a modular fashion with each component performing a highly specialized task. There are few hooks from one component into another to allow future versions of the components to be integrated into the system with minimal effort.

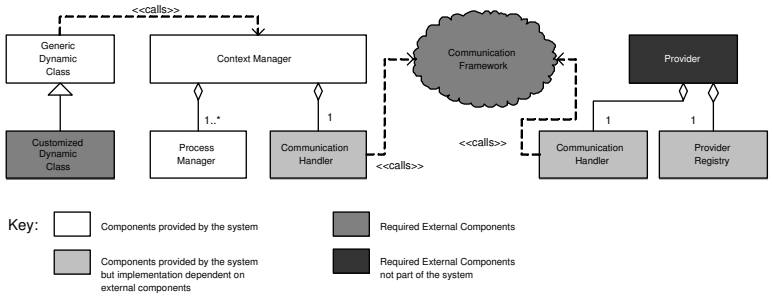

Fig. 2. Key Components of the System

Programmer Interface – The programmer interface to the middleware is designed to be easily integrated into any application that requires the use of context-sensitive binding. It is provided to the application developer in the form of a generic dynamic class that contains code required to build a customized dynamic class. The application programmer can extend this generic class and add methods of his own to create a customized dynamic class. The code inherited

from the generic class handles all calls to the context manager. On instantiation, the class automatically connects to the context manager to inform it of its presence. It then uses reflection to discover a list of its methods and their signature. It combines this list with user specified policies to create a request to the context manager to build an eligibility set for this dynamic class based on the method list and the attached policy. Once the list is built, the customized dynamic class can handle calls to any of its (dynamic class') methods that are made by the application. All calls made to the dynamic class are delegated to the context manager which transparently handles the processing of those calls and returns the result. The results that are returned by the context manager are propagated up to the application in the form of a simple method return.

Context Manager – The context manager is the main component of the system. There is a single instance of the context manager that runs on every host. It manages eligibility sets for every dynamic class on its host and mediates all communication between the program and the provider. When a dynamic class registers with the context manager, it generates and returns a globally unique `DynamicClassID`. Subsequently, when the context manager receives a method list and the associated policy from a dynamic class, it packages the method list and policy into a query and sends it to the provider registry. All providers receive this query and a subset send a responses back to the context manager. The context manager then builds an eligibility set of suitable providers. The volatility metric is used to choose the best provider. An entry is made in its mapping table. The entry contains a `DynamicClassID - ProviderID` pair. By looking up this table, the context manager can establish the relevant provider for any dynamic class that it is responsible for. At this point, the context manager creates a process manager for the dynamic class in question and passes to it the eligibility set. The process manager handles all subsequent communication between the dynamic class and its provider.

Process Manager – A process manager is created by the context manager for every dynamic class that it is responsible for. Once the context manager has defined an eligibility set for the dynamic class, it delegates the job of handling further communications to the process manager. The process manager receives the eligibility set in its constructor. It is also made aware of the best provider available. It instructs the communication handler to create a communication channel to that provider and waits till it receives an instruction. When a method call is made on a dynamic class, the call along with the relevant parameters are delegated to the context manager which then delegates them to the appropriate process manager. The process manager then communicates the call and its parameters to the provider using the channel created by the communication handler. It waits to receive the result from the provider and then propagates it up to the dynamic class by way of the context manager. In addition to this basic functionality, the process manager also caches partial results that are transmitted back periodically from the provider. In case the provider becomes unreachable, the process manager determines the next best provider and pushes the cached partial result to that provider with instructions to resume the computation from

when it was last cached. The process manager also keeps the eligibility set up to date by periodically asking the context manager to rebroadcast the initial request that was used to build the first version of the set.

Communication Handler – The Communication Handler is a component of the system whose implementation is dependent on the external communication framework used. An appropriate communication handler can be plugged into the system depending on what communication framework is used. The communication handler provides a layer of abstraction between the context-sensitive binding middleware and the communication middleware. It takes requests from either the context manager or any of the process managers and packages it in a manner that is legal for the communication framework and sends it to its destination. It performs a similar function on the provider side by abstracting all details of communication away from the provider.

Communication Framework – The communication framework handles all the communication between the interface and providers. This framework is not supplied as a part of the system. Instead, any existing system can be used to fulfil this functionality, if an appropriate communication handler is written. Given the environments that the context-sensitive middleware is designed for, coordination middleware for wireless ad hoc networks is preferred over other mechanisms to perform this task.

Provider Registry – The provider registry is a distributed data structure containing a set of *profiles* belonging to providers that can supply implementations for some set of methods. The profile of a provider consists of the list of methods and their signatures for which it has an implementation. The profile also contains external variables such as location of the provider, velocity of the provider, security clearances, etc. Each provider contains a local provider registry in which it places its own profile. The local registry merges with nearby provider registries to form a federated registry. By its structure, the provider registry contains profiles from reachable hosts only (because hosts that are unreachable cannot merge their registries) and hence any host that has a profile in the merged registry is guaranteed to be reachable.

Provider – The provider is an autonomous application that advertises its capabilities to hosts around itself. The provider advertises its profile in its local provider registry. As it encounters other providers, the registries merge. When the profile of a provider matches the requirements of some interface, it becomes part of the eligibility set. If a provider is chosen, method calls are delegated to it by sending messages across the communication framework. The provider locally executes the code for the appropriate method, packages the result and sends them back to the caller, again via the communication framework.

3.3 Anatomy of an Execution

Having examined the components of the system, we now illustrate how these components interact with each other. Fig. 3 shows a sample execution with the failure of one provider.

The execution starts at the application level when the application instantiates a customized dynamic class. When the constructor of this class is invoked, it first calls the constructor of its superclass, the generic dynamic class. The constructor of the generic dynamic class registers itself with the local context manager by calling a `register()` method on it. The `register()` method returns a unique `DynamicClassID` which is subsequently used to identify this particular dynamic class to the context manager. Once the `DynamicClassID` is obtained, the generic dynamic class uses reflection to discover the list of its methods. It then passes this list, along with the policy for the class to the context manager. The context manager packages the method list and policy in a message and sends the query to the provider registry. It should be noted that the registry shown in Fig. 3 represents a registry consisting of several merged registries.

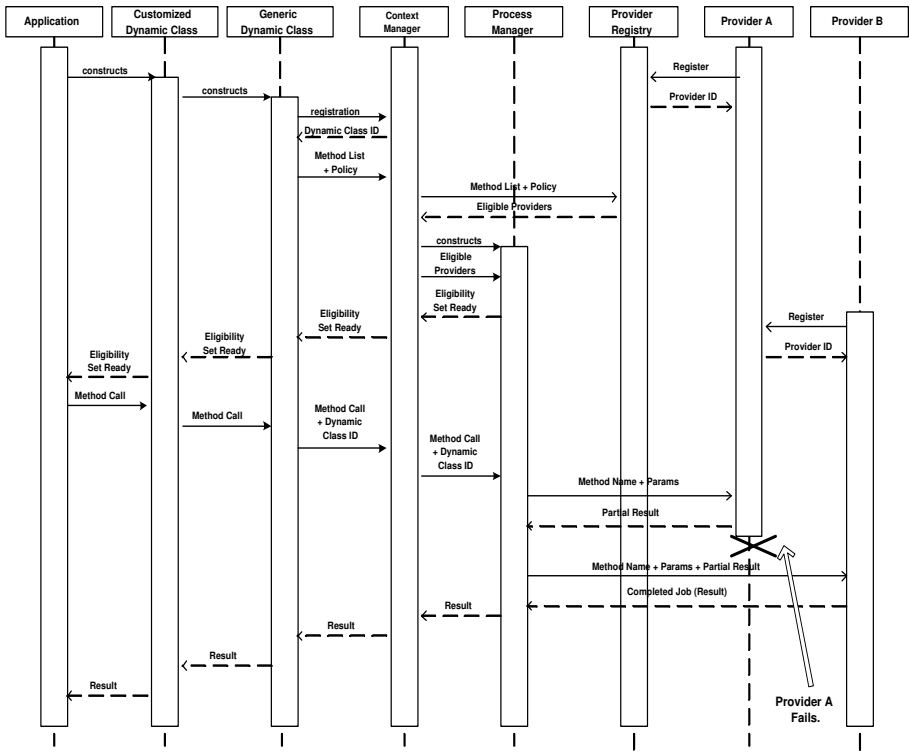

Fig. 3. Sample Execution of a context-aware application

All providers who are registered with the registry (i.e., their local registries are a part of the merged registry) and conform to the requirements set forth in the query reply to the query. The context manager uses these responses to build an eligibility set. It then creates a process manager and passes to it the

eligibility set. The process manager creates a link to the best provider in the set and returns an acknowledgement. This acknowledgement is propagated up to the application level via the context manager, generic dynamic class and the customized dynamic class. At this point, the application can start issuing method calls on the customized dynamic class. The figure shows how this request is delegated to a provider, the interactions that take place if a provider fails, and finally the returning of the result to the application.

4 Implementation

The implementation of the context-sensitive binding middleware is in Java. Due to constraints of space, we mention only selected details here.

The programmer interface, context manager, process manager, policy objects and constraints are all implemented as Java classes. For the purposes of a communication framework and a distributed provider registry, we use LIME [3], a middleware for physical and logical mobility. Our rationale for choosing LIME is that it is designed to be a communication framework for use in mobile ad hoc networks, which is the type of environment where we anticipate context-sensitive binding to be used to the greatest degree. The second reason was because of LIME's use of transiently shared tuplespaces. Tuplespaces are structures that can hold data. LIME uses the notion of a local tuplespace, which stores data local to a host. These local tuplespaces can be merged with tuplespaces belonging to reachable hosts to form a federated tuplespace. Such a data structure is exactly what is required to model a distributed provider registry and hence, we leverage off the power of LIME to implement them. The remainder of this subsection describes the programmer interface component in greater detail as this is the component that all programmers see. The knowledge of details of the other classes do not aid the programmer in writing applications hence we omit them here.

```
public abstract class Dynamic {

        public Dynamic(String subclassName, Constraints c)
                cm = ContextManager.getManager();
                myID = cm.registerClass()

                cm.buildWindow(myMethodList, c);
        }

        public Object invokeMethod(String methodName, Vector params) {
                cm.invokeMethod(myID, methodName, params);
        }

        public void dispose() {
                cm.destroyWindow(myID);
        }
}
```

Fig. 4. Code for Programmer Interface - The `Dynamic` class

The programmer interface is represented as a Java class called `Dynamic`. The `Dynamic` class has three basic methods. The constructor contains the code to register the class with the context manager, obtain a `DynamicClassID` and send the policy associated with this class to the context manager for the purposes of building the context. The `invokeMethod` method takes parameters from programmer defined methods and delegates them to the context manager. Finally, the `dispose()` method sends a message to the context manager when the class is no longer required. The methods of the dynamic class along with some sample pseudocode is shown in Fig. 4.

The Programmer Perspective

In this section, we illustrate how our middleware can be utilized to build a context-aware program. To do this, we revisit our example from Section 2 of the miner who has to walk through a mine to inspect it.

```
public class DynamicLightSwitch extends Dynamic{

        public DynamicLightSwitch(Constraints c) {
                super("DynamicLightSwitch", c);
        }

        public void on() {
                cm.invokeMethod(myID, "on", null);
        }

        public void off() {
                cm.invokeMethod(myID, "off", null);
        }
}
```

Fig. 5. The `DynamicLightSwitch` class

Recall the miner wished to have all the lights turn on as he approached them and turn off as he walked away from them. To incorporate such functionality, he would have to define a customized dynamic class that is tailored to his needs. We call this class `DynamicLightSwitch` and the code for this class is shown in Fig. 5.

Once the miner defines his custom dynamic class, all he has to do is write a simple java application that uses this class and invokes methods on it as necessary. The sample code for such an application is shown in Fig. 6. The first line of code defines the exogenous constraints, which is everything in the range 100 meters ahead and behind of the miner's position. The endogenous constraints require the provider to have an "on" and an "off" method. The program then instantiates a new instance of the `DynamicLightSwitch` and calls the `on()` method to turn the light on. After the inspection is complete, it calls the `off()` method to indicate that the miner is no longer interested in turning on lights and finally the `dispose()` method to indicate that the dynamic class will not be used anymore in the program. The potential for rapid development is made evident

```
public class ContextSensitiveSwitch{

        public static void main(String[] args) {
                Constraint locConstraint = new Constraint("Location",
                                                "Range", myLoc - 100, myLoc + 100);
                DynamicLightSwitch dls = new DynamicLightSwitch(locConstraint);
                dls.on();

                //Wait for duration of inspection

                dls.off();
                dls.dispose();
        }
}
```

Fig. 6. Sample Application for Mine Inspection

by the fact that it takes just 4 lines of code to design a complex context-aware application.

5 Discussion

There has been a significant effort directed towards designing tools to make applications context-sensitive. EgoSpaces [4] introduces the concept of a view, which is a subset of the context of a host. The Active Badge framework [5] allows queries to the host's current context and can generate notifications based on context changes. FieldNotes [6] introduces the ideas of multiple types of contexts with which it can tag data. The goal of all these tools is to give the programmer the power to develop context-aware applications. Examples of such applications include CyberGuide [7] and GUIDE [8] which are tour guide applications that use location information to update screens and the Stick-e Document [9] framework which triggers notes based on an associated context.

Context-sensitive binding is a novel perspective on exploiting context information to ease development of context-aware applications. It does this by decoupling the object interface from its realization and providing the framework to choose realizations for a given interface in a dynamic and adaptive manner. The middleware we developed implements the basic features that demonstrate the feasibility of the concept and illustrate the potential of the model. Context-sensitive binding abstracts details of dynamic context-aware binding and as such can be used in environments that are highly dynamic such as ad hoc wireless networks where disconnection is a common occurrence. Our middleware relies on a set of simplifying assumptions and initial design decisions. Relaxing these assumptions and design decisions exposes a wide range of research issues. In this section, we examine some of them and also highlight the applicability of this research in other interesting areas.

One of the key design decisions we took was in regard to matching the policy provided by a programmer to a profile advertised by a provider. Our current system uses an exact matching policy over the set of methods and constraints. This can be extremely restrictive. There is a need for a more flexible matching algorithm that can match sufficiently similar policies and profiles. Matching

based on semantics is one solution to this problem. However, that in turn raises issues of accurately describing semantics in a manner that is understandable by the system. For example, "on" and "off" may have the same semantics as "play" and "pause" in the context of a VCR.

Another important feature of this work is its ability to support composition. By changing the design of the system slightly, we can capture method implementations for a single class from multiple providers, in effect composing the capabilities of those providers under a single umbrella. The challenge is to ensure that these providers behave correctly as composed entities, i.e., they offer the correct functionality, do not deadlock, and maintain fairness. Issues regarding atomicity of such interactions in this model of programming also require further investigation. The development of a rich set of metrics that can be used to evaluate providers where the cause of context changes may differ is also an open area of research.

Context-sensitive binding can be applied to a variety of settings. Service oriented computing is a paradigm that is fast gaining popularity in wired settings in the form of web services [10]. Migrating such a framework to ad hoc networks requires a new perspective on the core concept. Context-sensitive bindings may hold the key to offering service oriented computing in ad hoc networks. Certain constructs offered by context-sensitive bindings are analogous, on a low level to those offered by a service oriented computing framework. For example, the process of finding an eligible provider is analogous to service discovery. Hence, context-sensitive binding can be used as low level middleware that abstracts details of disconnection and transient connectivity on top of which a middleware for service oriented computing for ad hoc networks can be built.

In addition to supporting service oriented computing [11] in ad hoc setting, context-sensitive binding can be used to foster novel patterns of interactions in ad hoc networks. Features such as periodic caching of state and support for pause-and-resume computing can support novel patterns of interactions which exhibit predictable, periodic connectivity between hosts. It can also be used as a possible means of composition of different applications by obtaining realizations from multiple remote objects and combining them under a common interface to yield a single object which is a conglomerate of selected capabilities from a set of remote objects.

Our future work in this area will focus on new concepts that give more power to the middleware. Examples of such concepts are finding providers based on semantics of an interface, obtaining realizations of a single interface from multiple providers, using the idea of network abstractions [12] to optimize the time required to find a service within a subset of the surrounding network, adaptable context management based on a changing set of metrics, among others. We also plan to focus on the mobility aspect of the concept and introduce mechanisms that allow intelligent decision making and forward looking predictions regarding which providers to use.

6 Conclusion

In this paper, we introduced the concept of context-sensitive binding. Context-sensitive binding separates the object interface from its implementation and transparently maintains the binding between the two based on context. The significance of this work rests with the introduction of constructs that transparently maintain and use context information to allow programmers to write simple code which leverages off a new form of dynamic binding designed for building complex, adaptive applications. We illustrated the feasibility of this concept by implementing a middleware supporting context-sensitive binding. We anticipate improving the model to give it more power, with the aim of further simplifying the programming task.

Acknowledgements

This research was supported by the Office of Naval Research under MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily represent the views of the research sponsors.

References

1. Lippman, R., Lajoie, J.: C++ Primer. Addison Wesley (1998)
2. Huang, Q., Julien, C., Roman, G.C.: Relying on safe distance to achieve partitionable group membership in ad hoc networks. Technical Report WUCS-02-35, Washington University, Department of Computer Science (2002)
3. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the 21st International Conference on Distributed Computing Systems. (2001) 524–533
4. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proceedings of the 10^{th} International Symposium on the Foundations of Software Engineering. (2002)
5. Harter, A., Hopper, A.: A distributed location system for the active office. IEEE Networks **8** (1994) 62–70
6. Ryan, N., Pascoe, J., Morse, D.: Fieldnote: A handheld information system for the field. In: 1^{st} Int'l. Workshop on TeloGeoProcessing. (1999)
7. Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: A mobile context-aware tour guide. ACM Wireless Networks **3** (1997)
8. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proc. of MobiCom, ACM Press (2000) 20–31
9. Brown, P.J.: The stick-e document: A framework for creating context-aware applications. In: Proc. of EP'96. (1996) 259–272
10. : W3c page for the web services activity. http://www.w3.org/2002/ws/ (2003)
11. Turner, M., Budgen, D., Brereton, P.: Turning software into a service. Computer (2003) 38–44
12. Roman, G.C., Julien, C., Huang, Q.: Network abstractions for context-aware mobile computing. In: Proceedings of 24th International Conference on Software Engineering. (2002) 363–373